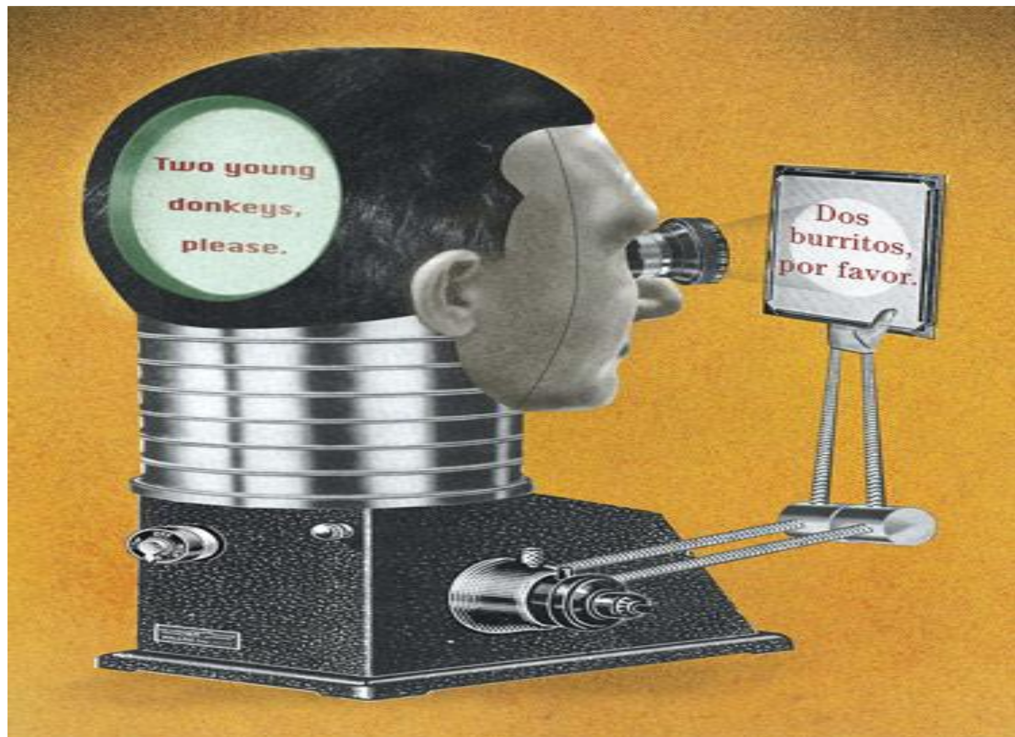
Institiúid Teicneolaíochta Cheatharlach

INSTITUTE of TECHNOLOGY
CARLOW
At the Heart of South Leinster

Occam & C++ Translator



Student Name: Shaoguang Miao

Student ID: C00131017

Supervisor: Joseph Kehoe

Table of Contents

I. **Introduction** 3

II. **Low Level Design** 4

1. Main Data structure Design 4
2. Lexer Design 6
3. Parser Design 11

III. **High Level Design** 24

1. Use Case Design 24
2. Contract 26
3. Modular Design 28
4. System Sequence diagram 31
5. Project Flow Chart 33
6. Class Diagram 34
7. Interface Design 35

IV. **Testing Design** 37

V. **Conclusion** 40

Reference 41

I. Introduction

This document will show the whole design par of the project. It includes low level design, hight level design and testing design. Because the Occam is a context free language, I can use Lexer and Parser way to do this project. However, the code written by people can be a lot of different situations, translate the whole project is impossible. First version of the project will translate some basic key words first. If there is enough time, I will continue to develop the next version.

In low level design, it introduces the main data structure, Lexer design and Parser design. Of course, the parser if most important part, because it will do the actually task of translating. In the parser part design, the algorithm for each import keywords was listed, like SEQ, PAR, and WHILE etc. In the first part of the low level design part, the main data structure has been given out. The stack is one of the most important data types, because the scope of each variable or keyword is significant thing in programming. Even there are a little bit problems about the scopes; the program will show the totally different result to the user. Thus first thing of the project is mark sure the scope of each word. In this process, the stack is the most useful data structure. Another important data type is linked list. Lexer will separate the whole Occam source code into a linked list, and then pass it to Parser. The entire thing for parser part is how to handle this linked list. How to define the type of the token is a problem for this project, because there is a lot way to do that. It is hard to say which one is the best. For this project, I select static const global variables to represent the type of the data. How to separate the Occam source code into tokens is the main task of the Lexer. In the Lexer part, I listed the algorithm about how to separate the code in to word, recognise the word and put them into the linked list.

In high level design, the main use case, use case diagram, operation contract, module diagram, system sequence diagram, class diagram and flow char was listed. It showed the whole structure and procedures of the project.

In testing design, there are numbers of test sample was listed for testing. In final project, those sample will be a part of basic test.

II. Low Level Design

1. Main Data structure Design

There are data structure I will use in this project, I will give the detail following .

```
struct token{
    int type;//Used to recored the type of the word.
    char *name;//Used to record the value of the  word.
    token *next;//Point to next node.
    token *pre;//Point to prvious node.
};
```

This structure will be used in the whole project. Because the Occam is a text-free language, I can use Lexer and Parser to handle translating. Every word from the Occam source code will be recognised as a token. This token should have data type, name, the pointer pointed to next token, and the pointer pointed to previse token. For every line of the Occam code, the Lexer will build a double linked list then pass it to Parser. Parser will translate it into new expressions.

```
struct stack{
    /*Using position to identify the position of indentation and number to indentify the number of variables in this scope. */
    int position;
    stack* next;
};
```

This structure will be used to record the position of beginning of each scope. Because the Occam uses indentation to mark the scope for each key word, I have to translate it into the curly bracket. Stack is the key of handling this problem. The Lexer and Parser part will give detail about how to use this structure.

```
static const int UNKNOWN=0;
static const int ASSIGN=1;
static const int QUESTION=2;
static const int EXCLAMATION=3;
static const int SKIP=4;
static const int STOP=5;
```

```
static const int SEQ=6;
static const int IF=7;
static const int WHILE=8;
static const int PAR=9;
static const int ALT=10;
static const int LR_BRACKET=11;
static const int RR_BRACKET=12;
static const int LC_BRACKET=13;
static const int RC_BRACKET=14;
static const int FOR       = 15;
static const int FROM = 16;
static const int NUMBER = 17;
static const int OPERATOR = 18;
static const int CHAN = 19;
static const int BOOL = 20;
static const int BYTE = 21;
static const int INT = 22;
static const int INT16 = 23;
static const int INT32 =  24;
static const int REAL32 = 25;
static const int REAL64 = 26;
static const int QUOTATION = 27;
static const int INDENTATION = 28;
static const int LA_BRACKET = 29;
static const int RA_BRACKET = 30;
static const int EQUAL = 31;
static const int END_NODE = 32;
static const int DECLARATION = 33;
static const int PUTINT = 34;
```

As mentioned before, there should have data type for each token. The list above is all the data type should be translated in this project. Using this way to define the data type is easier to understand and use.

2. Lexer Design

Lexer is actually a kind of word recognisor; it will get the token from the each line of the code. Making all tokens which in one line together in one linked list is the main task of the Lexer.

First of all, the Lexer has to know every character, number and operator, and then make the characters become new tokens. For this process the algorithm should like:

```
token *Lexer::sep_String(token *header,string word){

    while (index < word.length()){

        if(key_char == '/0'){ // Handle nothing in line but a enter
            continue;
        }
        else if (key_char == ':'){ // Handle ":=" or just ": "
            index++;
            key_char = word[index];
            if(key_char == '='){
                sub_str = ":=";
                index++;
            }
            else
                sub_str = ":";
        }
        else if(check_v_obj.check_Char(key_char) ){//If the key_char is a characters, do this

            index = next_variable_P(word,index_current); // Get next unchar's position.
            sub_str = word.substr(index_current,index - index_current);//Get the new substr.
            if (sub_str == "IF"){
                if_position = index - 2;
                if_Stack = push(if_Stack,if_position);//Add it into the if_stack;
            }
        }
        else if (check_v_obj.check_Number(key_char) ){//If the key_char is a number, do this.
            index = next_unnumber_Position(word,index_current);
            number_b = true;
            sub_str = word.substr(index_current,index - index_current);
        }
        else if (check_v_obj.check_Operator(key_char) ){ // Handle normal char
```

```
                    sub_str = key_char;//word.substr(index_current,index - index_current);
                    index++;
            }
            else if (check_v_obj.check_Brackets(key_char)){//Handle left bracket
                    sub_str = key_char;
                    index++;
            }
            else if(check_v_obj.check_Quotation(key_char)){ //Handle string or char
                    index = next_quotation_P(word,index_current);
                    sub_str = word.substr(index_current,index - index_current);
                    quotation_b = true;
                    if(sub_str[0] == '\''){
                            sub_str[0] = '\"';
                            sub_str[sub_str.length()] = '\"';
                    }
            }
            else if (key_char == ' ' || key_char == '\t'){//Check the space problem.

                    index = next_unspace_P(word,index_current);
                    continue;


            }
            else if(key_char =='\n')//Handle the only a enter in one line
                    continue;
        header = token_Linked(header,sub_str);// Add to linked list
        }
return header;
}
```

After the Lexer get one substring, it will pass it to the other function to handle it. In the new function, the string passed will be recognised. The new function will get the data_type and value, put them into the linked list.

```
token* Lexer::token_Linked(token *header,string words){
        Token token_obj;
        int w_type = 0;
        string w_name = words;//Store each sub_word of the  words.
        bool assignment_b = false; //Mark if this is assignment or not.

        if(words ==  ":="){
```

```
            w_type = token_obj.ASSIGN;
            assignment_b = true;
            add_first_b = true;
    }
    else if(words ==  "?"){
            w_type = token_obj.QUESTION;
            add_first_b = true;
    }
    else if(words ==  "!"){
            w_type = token_obj.EXCLAMATION;
            add_first_b = true;
    }
    else if(words ==  "SKIP"){
            w_type = token_obj.SKIP;
    }
    else if(words ==  "STOP"){
            w_type = token_obj.STOP;
    }
    else if(words ==  "SEQ"){
            w_type = token_obj.SEQ;
    }
    else if(words ==  "IF"){
            if_b = true;
            w_type = token_obj.IF;
    }
    else if(words ==  "WHILE"){
            w_type = token_obj.WHILE;
    }
    else if(words == "<"){
            w_type = token_obj.LA_BRACKET;
    }
    else if(words == ">"){
            w_type = token_obj.RA_BRACKET;
    }
    else if(words ==  "PAR"){
            w_type = token_obj.PAR;
    }
    else if(words ==  "ALT"){
            w_type = token_obj.ALT;
    }
```

```
else if (words == "{"){
        w_type = token_obj.LC_BRACKET;
}
else if (words == "}"){
        w_type = token_obj.RC_BRACKET;
}
else if(words == "="){
        w_type = token_obj.EQUAL;
}
else if (words == "FOR"){
        w_type = token_obj.FOR;
}
else if (words == "("){
        bracket_b = true;
        w_type = token_obj.LR_BRACKET;
}
else if (words == ")"){
        bracket_b = false;
        w_type = token_obj.RR_BRACKET;
}
else if (words == "CHAN"){
        w_type = token_obj.CHAN;
}
else if (words == "BOOL"){
        w_type = token_obj.BOOL;
        variable_b = true;
         data_type = w_type;
}
else if (words == "BYTE"){
        w_type = token_obj.BYTE;
        variable_b = true;
        data_type = w_type;
}
else if (words == "INT"){
        w_type = token_obj.INT;
        variable_b = true;
        data_type = w_type;
}
else if (words == "INT16"){
        w_type = token_obj.INT16;
```

```
            variable_b = true;
            data_type = w_type;
    }
    else if (words == "INT32"){
            w_type = token_obj.INT32;
            variable_b = true;
            data_type = w_type;
    }
    else if (words == "REAL32"){
            w_type = token_obj.REAL32;
            variable_b = true;
            data_type = w_type;
    }
    else if (words == "REAL64"){
            w_type = token_obj.REAL64;
            variable_b = true;
            data_type = w_type;
    }
    else if(words == "PUTINT"){
            w_type = token_obj.PUTINT;

    }
    else if(operator_b){
            w_type = token_obj.OPERATOR;
            operator_b = false;
    }
    else if(quotation_b)
            w_type = token_obj.QUESTION;
    else if(indentation_b){
            w_type = token_obj.INDENTATION;
            indentation_b = false;
    }
    else{
            if(data_type!=0)
                    w_type = data_type ;
            else
                    w_type = check_datatype(words);
    }

    token *newnode = new_Token(w_name,w_type);
```

```
        header = add_Token(header,newnode);

        return header;
}
```

After those two modules, the linked list produced by the Lexer will be passed into Parser part.

3. Parser Design

There are 5 constructer will be translated in the project, they are IF, WHILE, PAR, SEQ and CHANNEL. However they are the other few basic operator have to be translated too, like assignment, basic data type, etc. I will the detail of the data structure and algorithm about them following.

a. Basic Data types

They primitive Occam data type will be shown following (Table 1):

BOOL	Boolean values true and false. A boolean type.
BYTE	Integer values from 0 to 255. A byte type.
INT	Signed integer values represented in twos complement form using the word size most efficiently provided by the implementation. An integer type.
INT16	Signed integer values in the range –32768 to 32767, represented in twos complement form using 16 bits. An integer type.
INT32	Signed integer values in the range -2^{31} to $(2^{31} - 1)$, represented in twos complement form using 32 bits. An integer type.
INT64	Signed integer values in the range -2^{63} to $(2^{63} - 1)$, represented in twos complement form using 64 bits. An integer type.
REAL32	Floating point numbers stored using a sign bit, 8 bit exponent and 23 bit fraction in ANSI/IEEE Standard 754-1985 representation. The value is positive if the sign bit is 0, negative if the sign bit is 1. A real type. The magnitude of the value is: $(2^{(exponent-127)}) * 1.fraction$ if $0 < exponent$ and $exponent < 255$; $(2^{-126}) * 0.fraction$ if $exponent = 0$ and $fraction \neq 0$; 0 if $exponent = 0$ and $fraction = 0$
REAL64	Floating point numbers stored using a sign bit, 11 bit exponent and 52 bit fraction in ANSI/IEEE Standard 754-1985 representation. The value is positive if the sign bit is 0, negative if the sign bit is 1. A real type. The magnitude of the value is: $(2^{(exponent-1023)}) * 1.fraction$ if $0 < exponent$ and $exponent < 2047$; $(2^{-1022}) * 0.fraction$ if $exponent = 0$ and $fraction \neq 0$; 0 if $exponent = 0$ and $fraction = 0$

Table 1 [1]

Table 2 will showthe basic Occam type which will be translated to matched C++ data type. I will pick the some to them supported by C++ to translate, because they are necessary data type for any program.

OCCAM	C++
INT	**int**
INT 16	**short int**
INT 32	**int**
INT 64	**Not supported**
REAL 32	**float**
REAL 64	**double**
BYTE	**string**
BOOL	**bool**

Table 2

b. Operators

They are lots of different in operators between Occam and C++. The following will give the primitive operators out (Table 3).

Operator	Meaning	Operator	Meaning
+	addition	>>	shift right
-	subtraction	<<	shift left
*	multiplication	AND	boolean and
/	division	OR	boolean or
\ REM	remainder	NOT	boolean not
PLUS	modulo addition	=	equal
MINUS	modulo subtraction	<>	not equal
TIMES	modulo multiplication	<	less than
/\ BITAND	bitwise and	>	greater than
\/ BITOR	bitwise or	<=	less than or equal
><	bitwise exclusive or	>=	greater than or equal
~ BITNOT	bitwise not	AFTER	later than
SIZE	array size	BYTESIN	element size

Table 3 [1]

The matched operator in C++ will be given in Table 4. Those operators will be replaced immediately from Occam to C++. There are the other operators will can not be done as same as them will be given detail in the rest of this document.

<table>
<tr><th>Occam</th><th>C++</th></tr>
<tr><td>+</td><td>+</td></tr>
<tr><td>-</td><td>-</td></tr>
<tr><td>*</td><td>*</td></tr>
<tr><td>/</td><td>/</td></tr>
<tr><td>AND</td><td>&&</td></tr>
<tr><td>OR</td><td>||</td></tr>
<tr><td>NOT</td><td>!</td></tr>
</table>

=	==
<>	!=
<	<
>	>
<=	<=
>=	>=
/\	&
\/	|
>>	>>
<<	<<

Table 4

c. Assignments

In Occam, assignment is :=, = will be instead of it in C++. However there is a little difference rules beside they have different shape. About multi-assignment, in Occam like this:

a, b, c = d, e, f

However, in C++ it **not** like:

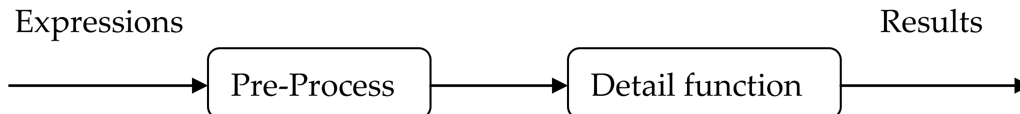
a = d, b = e, c = f

Because a list of expressions appearing to the right of the assignment symbol (:=) is evaluated in parallel, and then each value is assigned (in parallel) to corresponding variable of the list to the left of the symbol in Occam. But, the order of C++ is run them from left to right. Thus, the multi-assignment will be handled using multiple threads which will not be given detail here. How to change the format of them is also a problem. The Binary Tree will be used in whole project for handling lots of problems.

I will give just give the shape and theory here; the detail will give out in the other parts of this document.

For example:
Input is: a, b, c = 1,c+ 2, 3
They are using threads to be handled.
Process:

Before handle the assignment part, the Parser will do a pre-process first, because it will be easy to handle if all the assignments are in a same type of expression. In pre-pre-process, the mathematic expression (ie c + 2) will be translate into **type newvariable = c +2,** and then delete the c+2 from the linked list and add this newvariable into the linked instead. After that all of the assignment is like a same type of the expression. I want to build a new library to handle assignments. I will give a roughly idea about it below (Dig.1). However, I haven't decided how to hand this problem.



Dig.1

The binary tree will be traversed using NRL (Node, Right and Left). The detail will not be given in this document.

d. For Loop

Actually, there is no exactly For Loop in Occam. However, there is a replication in the other keywords. Thus, we also need make special module for the replication. The gramma of replication like this:

```
IF x = 1 FOR 5
    Do something
PAR x = 1 FOR 5
    Do something
SEQ x = 1 FOR 5
    Do something
```

Because it is between the other keywords and process expressions, it is very different with C++; I just need to separate for loop part and the IF (or PAR or SEQ) part as two parts. Here we just need to handle for part.

The algorithm of built for tree will show following;

```
string get_Replicator(token *token_list,string indentation){
    if(check_v_obj.check_Replicaotr(token_list)){
    //push the indentation number into the stack.
        Replicator = push_Stack(replicator,indentation_number);
        expression += indentation;//Add indentation
        //Get essiontial element from the linked list.
        string for_s = "FOR", equation = "=";
        token* equal=  get_Pointer(token_list, equation);//Get the pointer pointed to “=”
        token* for_p=  get_Pointer(token_list, for_s); //Get the pointer pointed to “FOR”
```

```
            //Get the variable which the controler of the for loop;
            /*the for loop in Occam is like
            SEQ I = 0 FOR 10
                Variable = base FOR mas_size
            */
            string variable = equal->pre->name;
            string max_size = for_p->next->name; //Get the limination;
            string base = equal->next->name;
            replicator_variables = add_V(replicator_variables, variable,token_obj.INT);


            expression += "for( int ";
            expression += variable;
            expression += "=";// index used in for loop
            expression += base;//start number
            expression += "; ";
            expression += variable; // index used in for loop
            expression += "<";
            expression += max_size; // max number
            expression += "; ";
            expression += variable;
            expression += "++){\n";


        }
        return expression;
    }
```

For handling this problem, we have to make sure what the scope of the FOR key word is. I will talk about the scope at the end of part 3.

e. IF Condition

I have mentioned it before, like what I said. There are two big differences between them. There is a pair of round brackets around the condition and square brackets around the expression and the IF key word and IF condition are in the separated line. We have to handle them separately.

The algorithm will show below:

```
string  if_Handler(token *token_list,string indentation){

        //Push the if position into if_position_stack

        if_position_stack = push_Stack(if_position_stack,indentation_number);

        //Get the replicaotr followed by if.

        replicator = get_Replicator(token_list,indentation);


        return replicator;

}

string condition_of_IF(token *token_list,string indentation){


        //handle the indentaion of if expression.

        if(if_condition_stack==NULL){ // This is not the first condition of if

                //Record the indentation numbers.

                if_condition_last = indentation_number;

                //Push the position of condition into stack.

                if_condition_stack = push_Stack(if_condition_stack,if_condition_last);

        }
```

```
        else{// This is the first condition of if
            /* there may be one nested if problem*/
            if(if_condition_last < indentation_number){  // handle nested_if

            if_condition_stack = push_Stack(if_condition_stack,indentation_number);
            if_condition_last = indentation_number;
            }
            else if(if_condition_last>=indentation_number){ // normal else if handler
                expression += indentation_if; //Make to C++ grammar
                expression += "}\n";//Make to C++ grammar
                expression += indentation_if; //Make to C++ grammar
                expression += "else if(";//Make to C++ grammar
            }
        }

        //Hanle the boolean expression part.
        expression += boolean_Expression(token_list);
        expression += "){\n";
        return expression;
    }
```

f. WHILE LOOP

Like IF, Occam use changing line to separate condition and expression, however, there is a pair of round bracket around condition. Occam uses indentation to represent the scope of variables and keywords; square bracket has been used to instead of them in C++. For handling this problem, we have to make sure what

the scope of the WHILE key word is. I will talk about the scope at the end of part3.

The main Algorithm for WHILE LOOP shown following:

```
string Parser::while_Handler(token *token_list,string indentation){
        //Record the position of while, put it into stack.
        while_stack = push_Stack(while_stack,indentation_number);
        temp = token_list ->next; // Next node
        expression+= boolean_Expression(temp); //Handle Boolean expressions
        return expression

}
```

g. PAR

PAR is the key word reserved for Occam complier which is for combining a number of processes which are performed concurrently. It means every process which is in this key word scope should be run parallel. This is the most important features of the Occam. Programmers can use the simplest code to do a very complex thing using Occam. For example:

```
index1 , index2 := 0,0
WHILE apple <>10
   PAR
        index1 = index1 +1
        index2 := index2 +1
```

The program will execute the expression1 and expression2 at same time. However, in normal programming languages, we have to build a thread for each expression. The Occam makes the parallel program easier to build. For handling

this problem, we have to make sure what the scope of the PAR key word is. I will talk about the scope at the end of part 3. Now I will give the algorithm of the PAR first:

```
Binary_tree *PAR_translate(){

        //Using for loop algorithm

        String expression = Check is there any for loop followed by PAR.

        Par_mark = true; //This is a global variable for mark the scope of the PAR

        //Record the position of the PAR.

        par_stack = push_Stack(par_stack,indentation_number);

}
```

h. SEQ

All the expression (process) in this key word scope should be produced one by one. It means every of the expression should be run under the sequence of the code.   For example;

```
index1 , index2 := 0,0

WHILE apple <>10

  PAR

      SEQ

            index1 = index1 +1

      SEQ

            index2 := index2 +1
```

Those two expression should be execute one by the other as same as usually program does. Of course there is the stack should be used to record the position of the first word of the SEQ to mark sure its scope. The algorithm will be shown below:

```
string Parser::sequence_Handler(token *token_list,string indentation){
	string replicator = ""; //Replicator expressions.

	seq_b = true;
	seq_inden = indentation_number;
	replicator = get_Replicator(token_list,indentation); //Handle the replicator.



	if(par_b){ //If the SEQ in the PAR, it will be handled in special way.
		par_seq_b =true; //Mark, the SEQ in the PAR.
		par_b = false; //Chang the property of the expression into SEQ.

		//Push the position into stack.
		seq_stack = push_Stack(seq_stack,indentation_number);
	}
	return replicator
}
```

Actually, if the SEQ is not in the PAR it will do nothing but followed by a for loop.

i. **Scope**

There is a huge different between Occam and C++ is the scope. The Occam uses the indentation to hanle a scope of the key word, but C++ use curly bracket to be instead of it. How ever in C++ there is also using indentation to make the code easy to understand. Thus in translated C++ scource file, the indentation should be keeped and curly bracket also should be add at same time. As mentationed before, whenever we use socpe key word (like IF, WHILE, PAR, SEQ), we should push the position into matched stack to make sure it scope. For same reason, when ever parser gets a new expression, it should check if it is out of some of the scope.

The algorithm shown below:

```
string Parser::indentation_Handle(token *token_list){
	token *current = token_list;
	string expression = "";
	string returned_in = current->name; // Get indentation from the linked list.

	if(token_list->type!= token_obj.END_NODE) //Check if it's the last expression.
		// Get the length of indentation.
		indentation_number =  returned_in.length();
	else
```

```
        indentation_number = 0;

    //********************** Check if Scope  ****************************
    if(if_position_stack!= NULL){
            expression = if_Scope(token_list);

    }

    //***********Check Replicator  Scope*****************************************

    if(replicator!=NULL){
        expression = replicator_Scope(token_list);

    }

    //****************************Check WHILE Scope***********************************
    if(while_stack != NULL){
        expression = while_Scope(token_list);

    }

    //*************************Check PAR Scope*************************************
    if(par_stack!=NULL)
        par_Scope(token_list);


    //************************Check SEQ Scope********************************
    if(seq_stack!= NULL)
        seq_Scope(token_list,returned_in);

    return returned_in;
}
```

III. High Level Design

1. Use Case Design

1.1 Use Case: Open File

Actor: User

Description: User clicks the open file button (or open file in menu), the software will show the dialog. User selects the file which will be opened and click submit button. The software will read the file and put the content of the file into left text box of the software. User may cancel the file it select, the software will close the open file dialog.

1.2 Use Case: Save File

Actor: User

Description: After user modified the content of the Occam source file, user clicks the save button (or save option in the menu), the software will save it into the file which is opened by user before. User also could select Save As in the menu, the software will show a save file dialog. User could fill the name and click the submit button to save it or click cancel button to cancel it. After the user click the button, the software will close the save file dialog.

1.3 Use Case: Export File

Actor: User

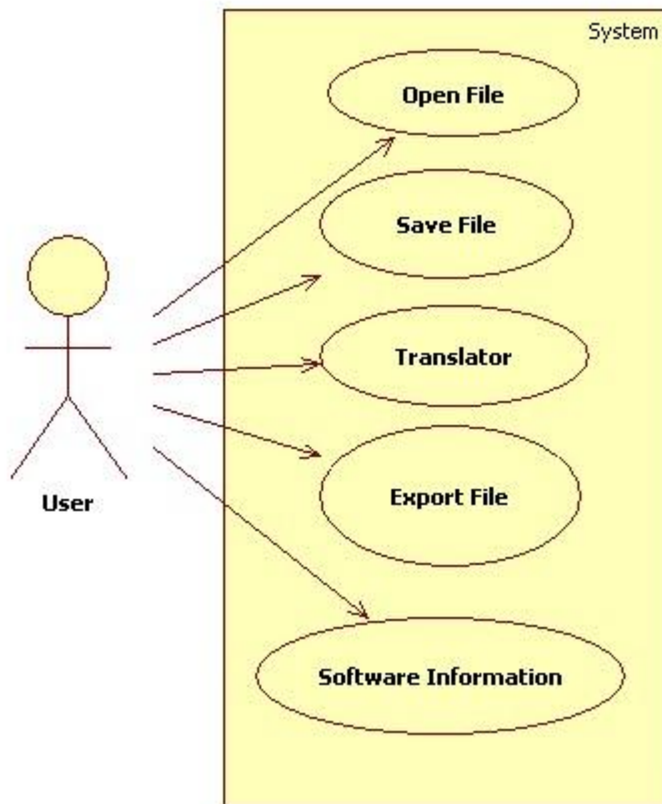
Description: After user translated the Occam source code into C++ source code, user click the Export File in the menu and the software will show an Export file dialog. After user fills the name of the file click submit button to save it or cancel button to cancel it. The software closes the Export file dialog.

1.4 Use Case: Translate

Actor: User

Description: After user click the translate button (or translate option in the menu), the software translate this Occam source code into C++ source code. And then show the result into the right text box area.

1.5 Use Case Diagram



2. Contract

Operation: open_file(string : path)

Cross References: Use Case: Open File

Precondition: User selected the file from the open file dialog

Post condition:

- The content of the file was shown in the left text box area.
- *path* was associated with the path parameters.
- a open file dialog instance was created.

Operation: Save As

Cross References: Use Case: Save File

Precondition: User fills the path of the new file

Post condition:

- A save file dialog instance was created.
- The content of the left text box area was saved in the special path.
- A new file was created

Operation: Export File

Cross Reference: Use Case: Export File

Precondition: User fills the path of the new file

Post condition:

- A Export file dialog instance was created.
- A new file was created.
- The content of the right text box area was saved in the special path.

Operation: Translate

Cross Reference: Use Case: Translate

Precondition: There is a file has been open in the left text box.

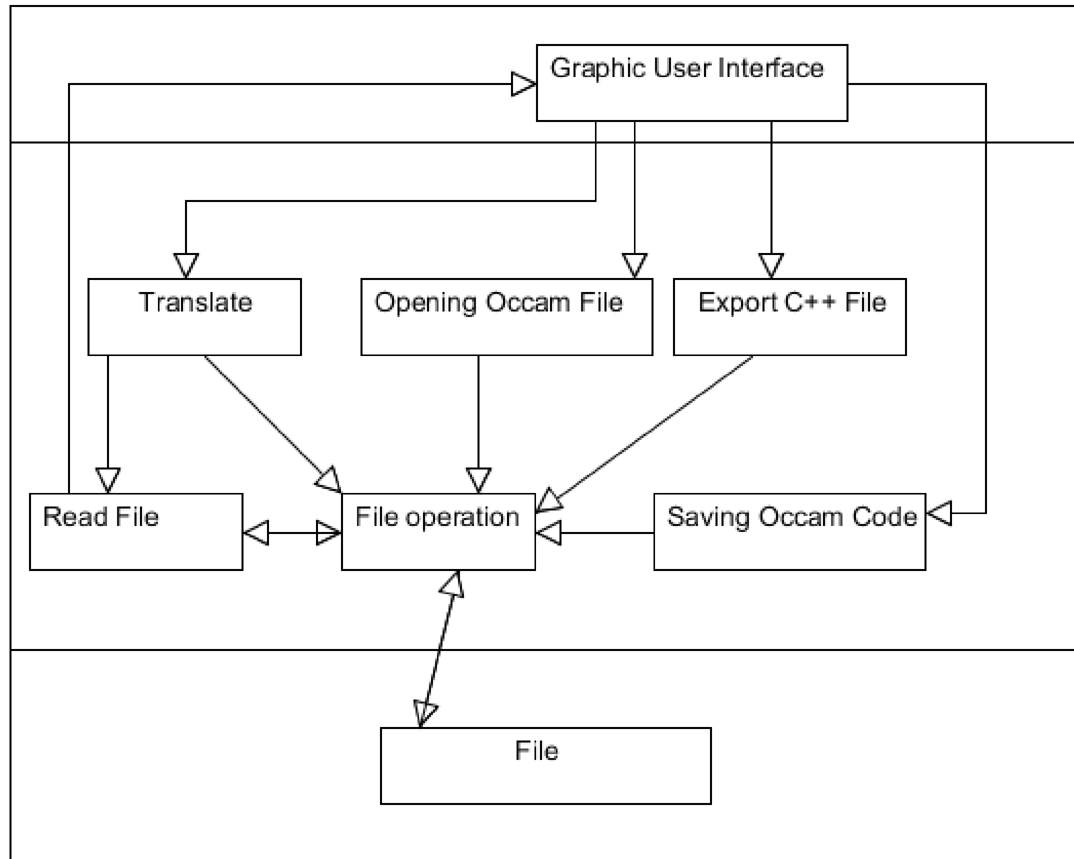
User clicked the translate button.

Post condition:

- Three template files were created.
- Lexer class instance was created.
- Parser class instance was created.
- The translated result was shown in the right text box area.

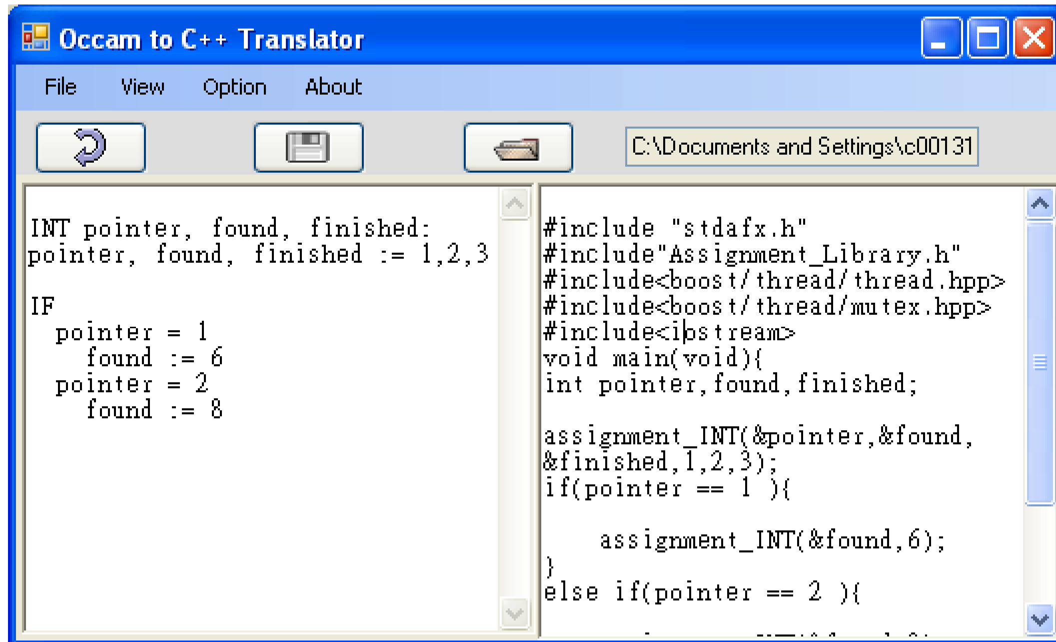
3. Modular Design

a. Modular Design Diagram



b. Graphic User Interface:

User will control the software through out of Graphic User Interface. It is also the only thing the user can know about the software, thus I try my best to make it looking better. The main window showed below (Fig.1). Actually, it is followed by standard rules to make it easy to use for user.



c. Open Occam File:

After the user clicked the open file button, this module will be used. An open file dialog will be showed on the top of main window; user could select the file which he wants to open. After the use submits the file, the content of the file selected will be shown in the left text box area. This must be the first step of the whole project. Translate and save button can not be used without opening any file before.

d. Save Occam File:

After the user modified the content of the Occam source file, the user could use this module to save a file into directory. The user could save the current file or save it as the other new file which will be created by the software.

e. Export C++ File:

This is exactly like save as option, the different is it will export a C++ file. This module can not be used unless the user has translated Occam file to C++ file.

f. **Translation**

This is the core module of this project. There are two parts in this module. First of all, the Lexer will recognise every string from the Occam source and put it in a linked list and pass the linked list to Parser which is the other part of this module. Parser will translated the linked list and write the result into a template file. After all the translating, the Lexer will make the entire of all template files together as a new C++ files.

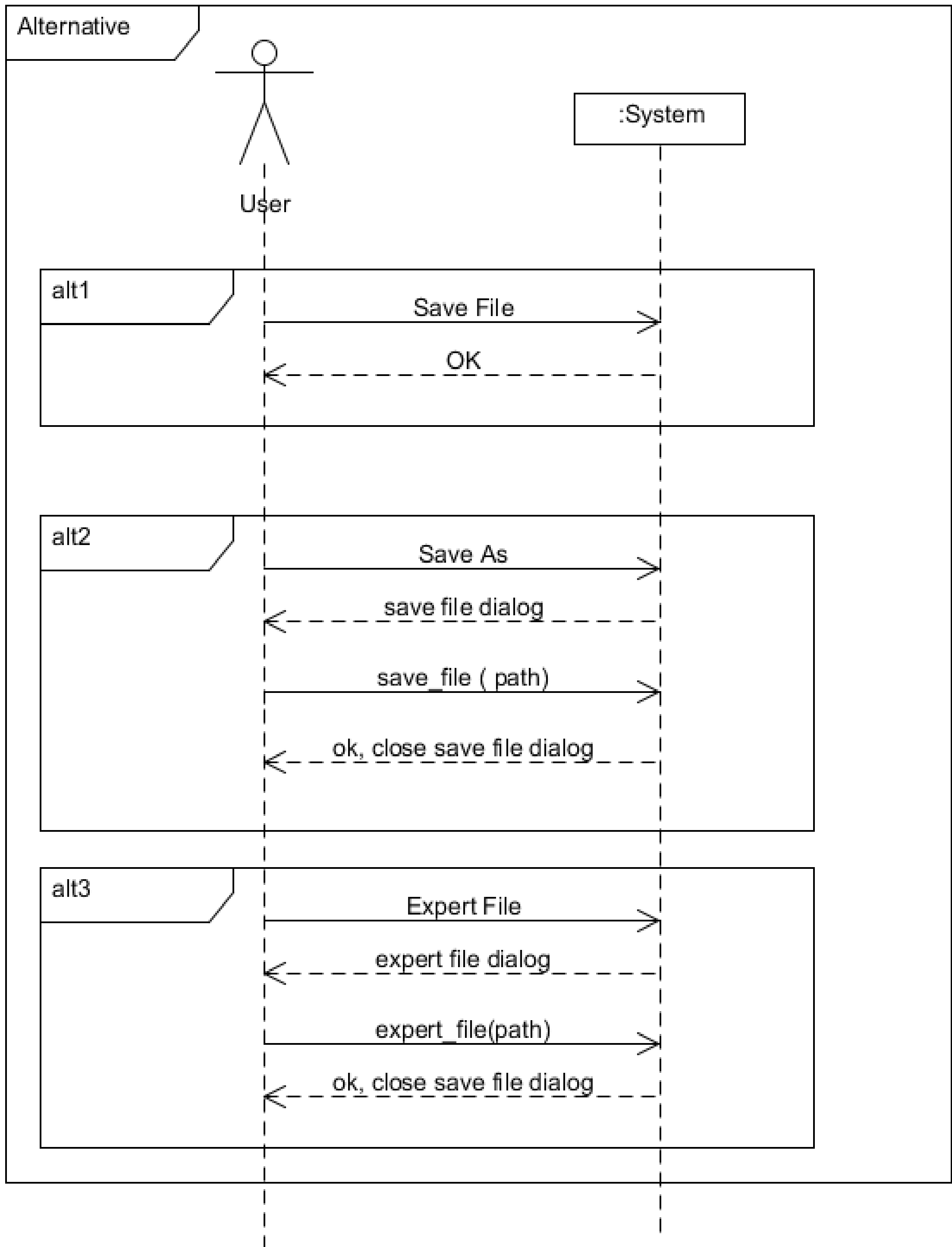
g. **File Operations**

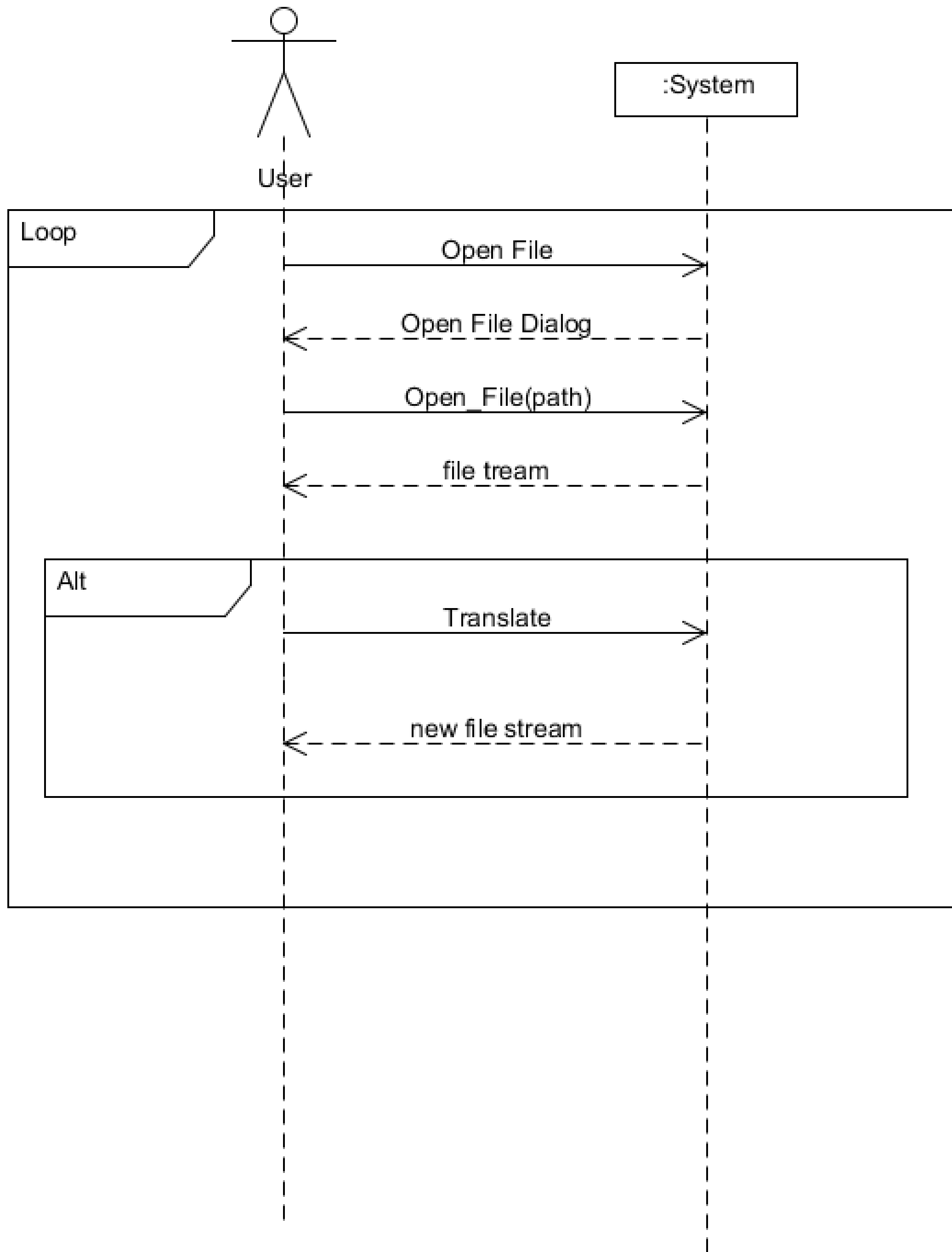
This file used the I/O stream to read from and write to the files. Actually, I did not create an individual class for this module, but it is a special module of the project. Whatever it is GUI, Lexer or Parser; this part will be used in it. All of the operations to the files are included in this module.

h. **File**

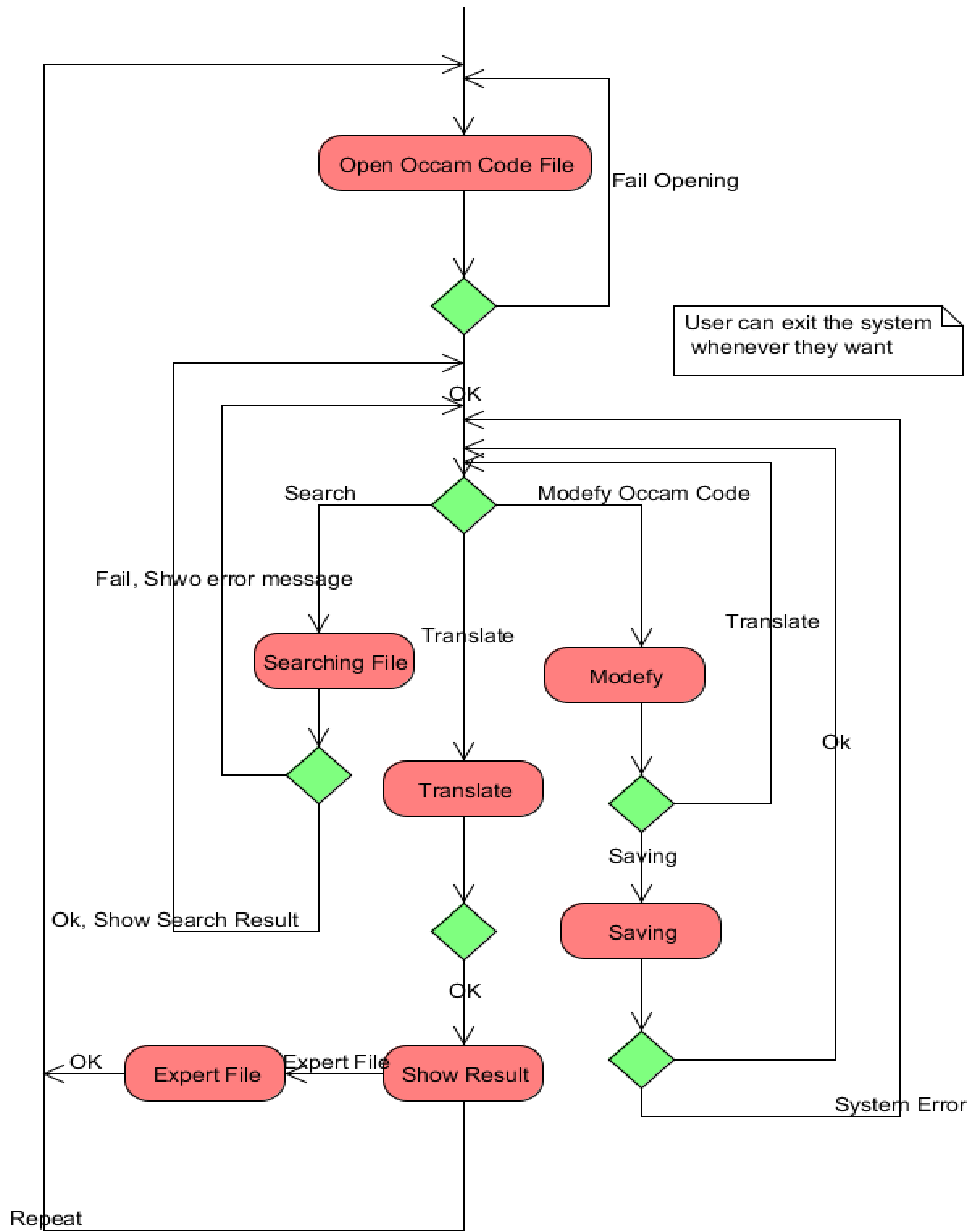
This module is the memory module. This project used three template file for store the useful information. All of the template files should belong to this module.

4. System Sequence diagram

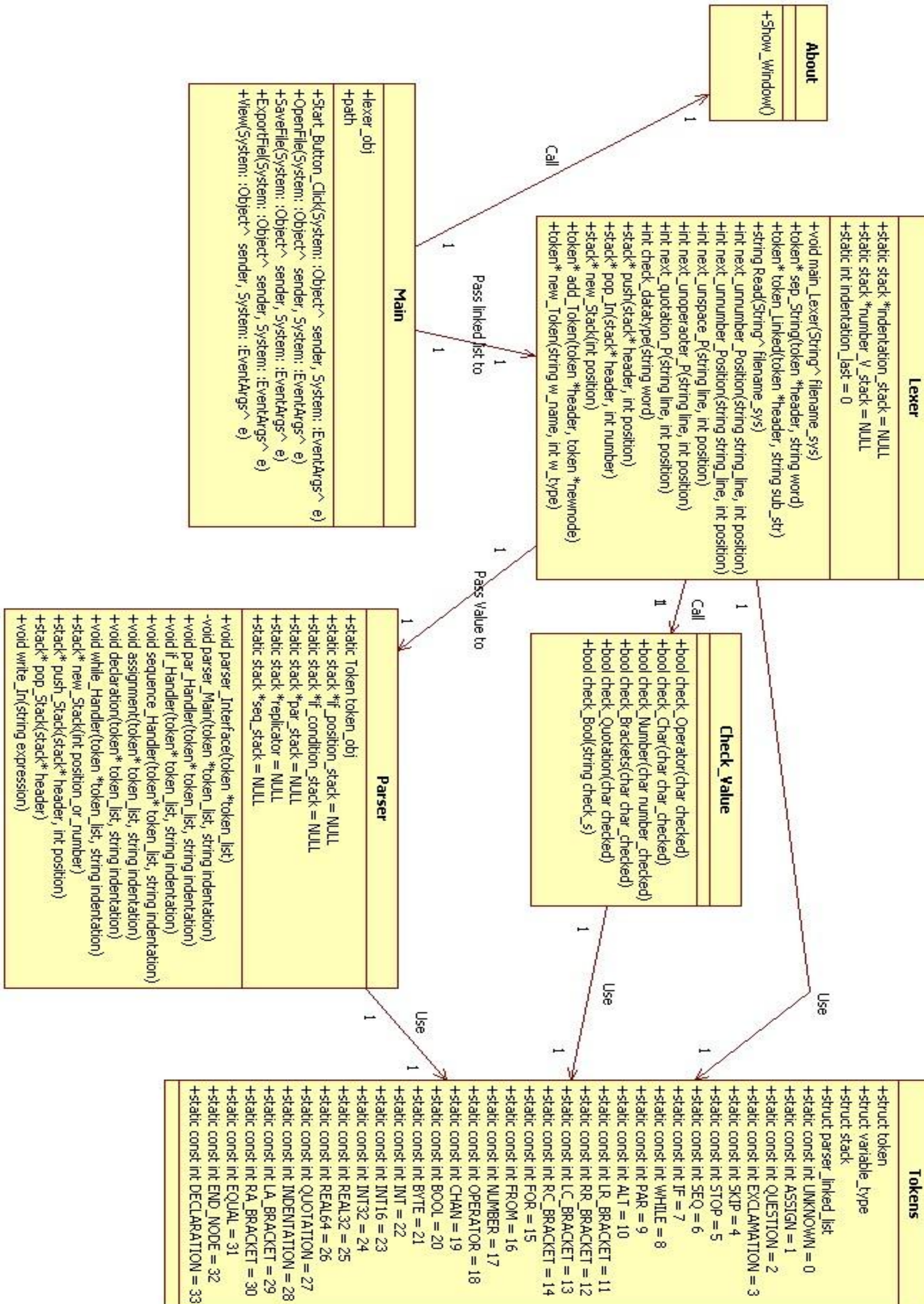



User
:System
Loop
Open File
Open File Dialog
Open_File(path)
file tream
Alt
Translate
new file stream

5. Project Flow Chart



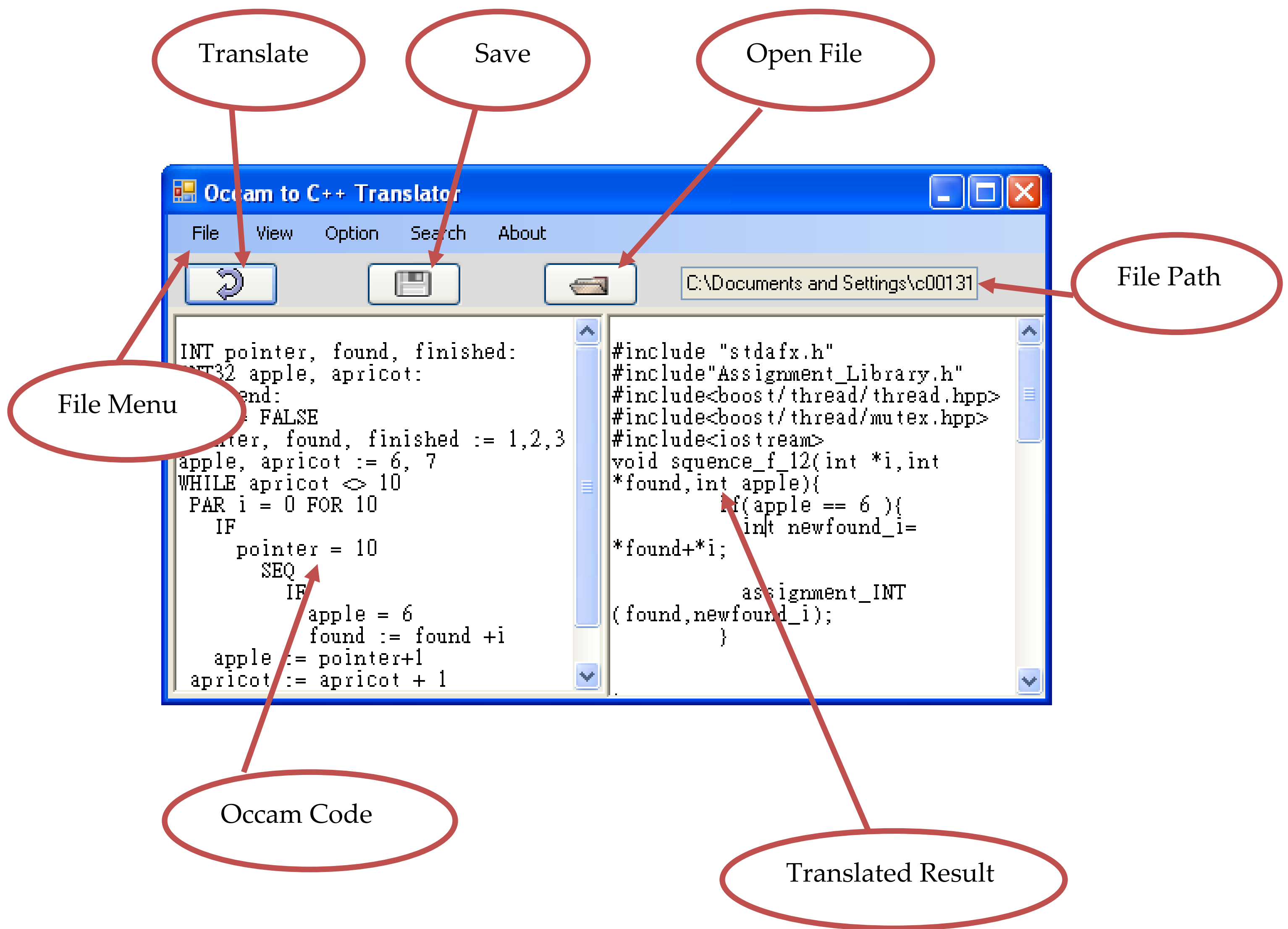
6. Class Diagram



7. Interface Design

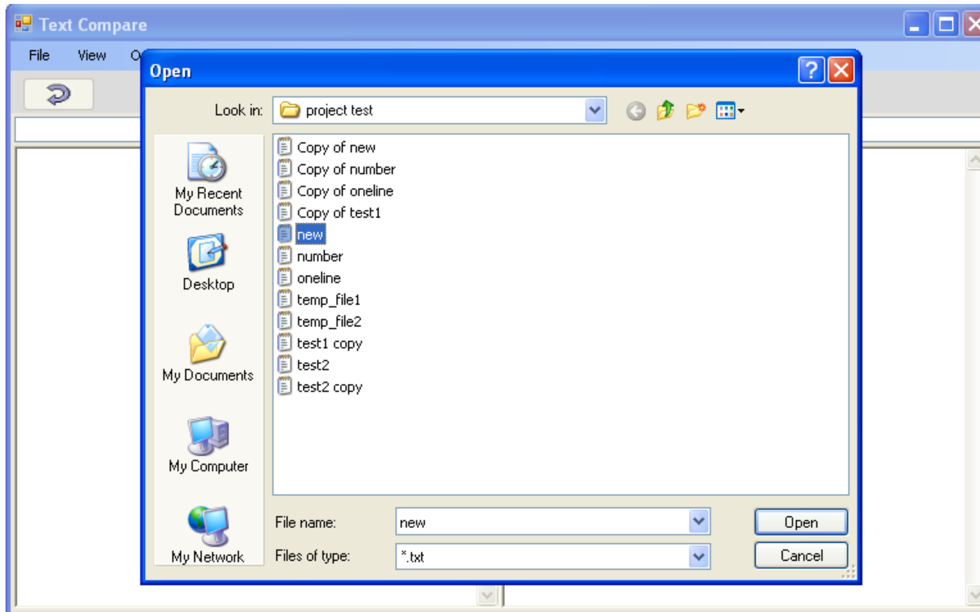
Interface is the other import part for the software. Here I will show the main part of the interface.

The main interface of this project showed below (Dig.2):



Dig.2

After user click open button, the open file dialog will be shown below (Dig. 3):



Dig. 3

After user click Help button, the about button will be show:

Dig. 3

IV. Testing Design

How to test this program is a big problem for this project, because the code written by people. There are too many situations in their coding. The more tests I will do, the better my project it is. For each key word there should be one testing program shown below:

For simple assignment:

```
INT32 apple, apricot:

apple := 2

apricot := 3

apple := apple + 1
```

For multi-assignment:

```
INT32 apple, apricot:

apple := 2

apricot := 3

apple,apricot := apricot, apple
```

For IF statement:

```
INT pointer, found, finished:

pointer, found, finished := 1,2,3



IF

  pointer = 1

        found := 6

  pointer = 2

found := 8
```

For Nested IF statement:

```
INT pointer, found, finished:

pointer, found, finished := 1,2,3



IF

    pointer = 1

        IF

          finished = 2

              found := 6

    pointer = 2

        found := 7
```

For replicator:

```
INT pointer, found, finished:

pointer, found, finished := 1,2,3



SEQ i = 0 FOR 10

     pointer := i + pointer

     finished := i * finished
```

For WHILE:

```
INT pointer, found, finished:

pointer, found, finished := 1,2,3

WHILE pointer <> 10

     pointer := pointer+1

     found := found + 1
```

For WHILE_IF

```
INT32 apple, apricot:

apple, apricot := 6, 7

WHILE apricot <> 10

    IF

      apricot <> 10

        IF

            apple = 6

                apple := apple+7

      apricot := apricot + 1
```

For Nested Loop:

```
INT32 apple, apricot:

apple, apricot := 6, 2

WHILE apricot < 3

    SEQ i = 0 FOR 10

        apple := apple +1

    apricot := apricot + 1
```

For PAR:

```
INT32 apple, apricot:

apple := 1

apricot := 2

PAR i = 0 FOR 10

  apple := apple + 1

  apricot := apricot + 1
```

V. Conclusion

The design could be the most important part of the whole project. After it, I will do the project followed by this document as close as possible to make sure I am doing it in a right way.

I have defined the data type and keyword which this project will handle in the low level design. In low level design part, the details of main data structures which will be widely used are also mentioned. Several import algorithms of Lexer and Parser are given out in the low level design.

In the hight level design, there are lot of UML diagram, like System Sequence Diagram, Class Diagram, Module Design Diagram, and Flow Chart Diagram etc. Those diagrams clearly showed the structure of the project.

Test is always the difficult of every project. It is impossible to make software fully correct. However, the more test I do, the better the project it is. There is lots of testing program showed in the testing design part. At least, the project should translate all of them in to fully correct result.

Reference

[1] Geraint Jones, 2001

http://www.comlab.ox.ac.uk/people/geraint.jones/publications/book/Pio1/